# Premise

The premise of CodeFirstMetadata is that there exists a simplest form of any problem. This code can be used to create the final or “real” code that can communicate the full intent to the humans. In many cases, the simplest form generated a number of artifacts that represent the form in different parts of the application.

This simplest form is expected to often be the single point of truth for all, or part of the metadata for the application. However, there is no reason to avoid metadata extraction steps creating these simplest artifacts as a step in a larger process.

# Goal

The goal of CodeFirstMetadata is to simplify mapping between simple code artifacts and strongly typed metadata. All currently imagined cases use this strongly typed metadata in further generative steps. This simplification is both a guide in defining the simplest definition and in mapping data.

# Definitions

***Real code:*** The C# code that is used by the compiler to create IL. This is also the code that other code in the application interacts with and the code generally used for debugging.

***Simplest form:*** Code that *parallels* the real code containing the simplest representation of the problem.

***Representational form:*** Code that contains a simplest representation of the problem that maps to, but does not parallel the real form.

The key difference between simplest form and representational form is whether the form parallels the real code or is orthogonal to it. In current samples, this also reflects whether the real code contains a method or property for each item in the form, which is the case with the simplest form. In current samples, the representational form works best when the real code always has the same properties and methods and the form is supplying the data, including the code to run, to create this standard pattern.

This is a somewhat arbitrary distinction because what appears simplest in relation to one artifact may appear representational when compared to another artifact.

However, the distinction remains because of current technical distinctions. One form may win, or the technical separation may disappear.

Some problems are more easily described with simplest parallel and some with representational.

# Examples

## Simplest Parallel Form

Given the code

using System;

using System.Diagnostics.Tracing;

namespace ConsoleRunT4Example

{

[EventSource(Name = "ConsoleRunT4Example-Normal")]

public sealed partial class Normal : EventSource

{

#region Standard class stuff

// Private constructor blocks direct instantiation of class

private Normal() {}

// Readonly access to cached, lazily created singleton instance

private static readonly Lazy<Normal> \_lazyLog =

new Lazy<Normal>(() => new Normal());

public static Normal Log

{

get { return \_lazyLog.Value; }

}

// Readonly access to private cached, lazy singleton inner class instance

private static readonly Lazy<Normal> \_lazyInnerlog =

new Lazy<Normal>(() => new Normal());

private static Normal innerLog

{

get { return \_lazyInnerlog.Value; }

}

#endregion

#region Your trace event methods

[Event(1)]

public void Message(String Message)

{

if (IsEnabled()) WriteEvent(1, Message);

}

[Event(2)]

public void AccessByPrimaryKey(Int32 PrimaryKey)

{

if (IsEnabled()) WriteEvent(3, PrimaryKey);

}

#endregion

}

}

The simplest, and parallel version is:

namespace ConsoleRunT4Example

{

[SemanticLog()]

public class Normal

{

public void Message(string Message) { }

public void AccessByPrimaryKey(int PrimaryKey) { }

}

}

Comparing these two code fragments shows that the two methods of the simplest form exist in the real code. Boatloads of gunk is removed, but the simplest parallel version is an abbreviated version of the real code.

## Representational Form

Some problems are inherently more complex, and a representational form can help guide the programmer to create the complex result. This sample also includes code, but to be clear, code can exist in either approach.

using System;

using System.Collections.Generic;

using System.Collections.Immutable;

using System.Linq;

using System.Threading;

using Microsoft.CodeAnalysis;

using Microsoft.CodeAnalysis.Diagnostics;

using Microsoft.CodeAnalysis.CSharp;

using Microsoft.CodeAnalysis.CSharp.Syntax;

namespace DiagnosticAndCodeFix

{

// TODO: Consider implementing other interfaces that implement IDiagnosticAnalyzer instead of or in addition to ISymbolAnalyzer

[DiagnosticAnalyzer(LanguageNames.CSharp)]

public class DiagnosticAnalyzer : ISyntaxNodeAnalyzer<SyntaxKind>

{

public const string DiagnosticId = "KADGEN1002";

internal const string Description = "Insiste on curly braces";

internal const string MessageFormat = "'{0}' needs a brace";

internal const string Category = "Style";

internal static DiagnosticDescriptor Rule = new DiagnosticDescriptor(DiagnosticId,

Description, MessageFormat, Category, DiagnosticSeverity.Error, true);

public ImmutableArray<DiagnosticDescriptor> SupportedDiagnostics { get { return ImmutableArray.Create(Rule); } }

public ImmutableArray<SyntaxKind> SyntaxKindsOfInterest

{

get

{

return ImmutableArray.Create(SyntaxKind.IfStatement, SyntaxKind.ElseClause );

}

}

public void AnalyzeNode(SyntaxNode node, SemanticModel semanticModel,

Action<Diagnostic> addDiagnostic, AnalyzerOptions options, CancellationToken cancellationToken)

{

var ifStatement = node as IfStatementSyntax;

if (ifStatement?.Statement != null

&& !ifStatement.Statement.IsKind(SyntaxKind.Block))

{

Report(ifStatement.IfKeyword.GetLocation(), "if statement", addDiagnostic);

}

var elseClause = node as ElseClauseSyntax ;

if (elseClause?.Statement != null

&& !elseClause.Statement.IsKind(SyntaxKind.Block))

{

Report(elseClause.ElseKeyword.GetLocation(), "if statement", addDiagnostic);

}

}

private void Report(Location location, string v, Action<Diagnostic> addDiagnostic)

{

var diagnostic = Diagnostic.Create(Rule, location, v);

addDiagnostic(diagnostic);

}

}

}

Because this is a somewhat complex domain that’s generally unfamiliar, I want to offer a lot of hints to the programmer - I’m doing this in the base class:

using CodeFirstAnalyzer;

using Microsoft.CodeAnalysis.CSharp.Syntax;

using Microsoft.CodeAnalysis;

using Microsoft.CodeAnalysis.CSharp;

namespace KathleensAnalyzer

{

class IfBraceFixer : DiagnosticBase

{

public IfBraceFixer()

{

Id = "KADGEN1001";

Description = "Needs braces";

MessageFormat = "{0} needs braces";

Category = "Style";

Analyze<IfStatementSyntax>(

condition: x => !x.Statement.IsKind(SyntaxKind.Block),

location: x => x.IfKeyword.GetLocation(),

messageArg: "if statement");

SimpleFix<IfStatementSyntax>(

newNode: x => x.WithStatement(

SyntaxFactory.Block(x.Statement)),

messageArg: "Add braces");

}

public class ElseBraceAnalyzer : KindAnalayzer<ElseClauseSyntax>

{

public override void AnalyzeNode(ElseClauseSyntax node)

{

if (!node.Statement.IsKind(SyntaxKind.Block))

{

Report(node.ElseKeyword, "else clause");

}

}

}

}

}

The code for the simple fix is in a different file, not included here.

Two diagnostics are described, each using a different format.

The IfBraceAnalyzer is described in the simple form of supplying